Smart Contract Security Audit Report

Zircuit ZRC Token

1. Contents

2. General Information

This report contains information about the results of the security audit of the Zircuit (hereafter referred to as "Customer") smart contracts, conducted by Decurity in the period from 07/29/2024 to 07/31/2024.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the contracts in the following repository: https://github.com/zircuit-labs/zrc-token. Initial review was done for the commit 16700c and the re-testing was done for the commit 37cc78.

The following contracts have been tested:

- src/ZRC.sol
- src/ZRCL2.sol

2.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token service owner, a contract). The centralization risks have not been considered upon the request of the Customer.

The main possible threat actors are:

- Token holders,
- Protocol owner,

- Optimism bridge contract.

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using the limited resources.

3. Summary

As a result of this work, we have not discovered any exploitable security issues.

The other suggestions included fixing the low-risk issues and some best practices (see Security Process Improvement).

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of August 7, 2024.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Ownable is used instead of Ownable2Step	src/ZRC.sol, src/ZRCL2.sol	**Low**	Acknowledged
Internally not invoked functions are marked as public	src/ZRCL2.sol	**Info**	Acknowledged
State variable declaration should include comments	src/ZRC.sol, src/ZRCL2.sol	**Info**	Acknowledged
Named parameters are not used in mappings	src/ZRC.sol, src/ZRCL2.sol	**Info**	Acknowledged
`onlyOwner` functions not accessible if owner renounces ownership	src/ZRC.sol, src/ZRCL2.sol	**Info**	Acknowledged
Floating pragma is used	src/ZRC.sol, src/ZRCL2.sol	**Info**	Acknowledged
For loop gas optimization	src/ZRC.sol, src/ZRCL2.sol	**Info**	Acknowledged
Deployment script uses not the expected private key		**Info**	Acknowledged

Issue	Contract	Risk Level	Status
Typo in comments	src/ZRC.sol src/ZRCL2.sol	**Info**	Fixed
Require is used instead of safeTransferFrom	utils/BatchTransfer.sol	**Info**	Acknowledged
Functions lack sanity checks	src/ZRCL2.sol	**Info**	Acknowledged

4. General Recommendations

This section contains general recommendations on how to improve overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. Ownable is used instead of Ownable2Step

Risk Level: Low

Status: Customer: the owner here does not really have much privilege

Contracts:

- src/ZRC.sol,
- src/ZRCL2.sol

Description:

The `transferOwnership` function is used to change ownership from `Ownable.sol`. The owner may accidentally specify a non-active address and lose access. `Ownable2Step.sol` is more secure due to a 2-stage ownership transfer.

Remediation:

We recommend using the `Ownable2Step` contract from OZ (Ownable2Step.sol) instead in both of ZRC and ZRCL2 contracts.

5.2. Internally not invoked functions are marked as public

Risk Level: Info

Status: Acknowledged

Contracts:

- src/ZRCL2.sol

Description:

Contracts are allowed to override their parents' functions and change the visibility from external to public.

```
File: src/ZRCL2.sol

135:     function nonces(address user) public view override(Nonces,
ERC20Permit) returns (uint256) {
```

Remediation:

Consider changing visibility to external.

5.3. State variable declaration should include comments

Risk Level: Info

Status: Acknowledged

Contracts:

- src/ZRC.sol,
- src/ZRCL2.sol

Description:

State variables should be commented to explain their purpose.

```
File: src/ZRC.sol

9: contract ZRC is Ownable, ERC20Permit {
10:     bool public locked;

File: src/ZRCL2.sol

13: contract ZRCL2 is IOptimismMintableERC20, Ownable, ERC20Permit, ERC20Votes
{
14:     bool public locked;

15:     address public immutable BRIDGE;
16:     address public immutable REMOTE_TOKEN;
```

Remediation:

Consider adding NatSpec for state variables.

5.4. Named parameters are not used in mappings

Risk Level: Info

Status: Acknowledged

Contracts:

- src/ZRC.sol,

- src/ZRCL2.sol

Description:

Named paremeterns are not used in several mappings:

```
File: src/ZRCL2.sol

19:     mapping(address => bool) public allowedSenders;

21:     mapping(address => bool) public allowedReceivers;

File: src/ZRC.sol

19:     mapping(address => bool) public allowedSenders;

21:     mapping(address => bool) public allowedReceivers;
```

Remediation:

Use named parameters: `mapping(address allowed => bool status) public allowedSenders;`.

5.5. onlyOwner functions not accessible if owner renounces ownership

Risk Level: Info

Status: Acknowledged

Contracts:

- src/ZRC.sol,
- src/ZRCL2.sol

Description:

The `owner` is able to perform certain privileged activities, but it's possible to set the owner to `address(0)`. This can represent a certain risk if the ownership is renounced for any other reason than by design. Renouncing ownership will leave the contract without an owner, therefore limiting any functionality that needs authority.

```
File: src/ZRCL2.sol

143:     function unlock() external onlyOwner {

154:     function setAllowedSenders(address[] memory _allowedSenders, bool
allow) external onlyOwner {

166:     function setAllowedReceivers(address[] memory _allowedReceivers, bool
allow) external onlyOwner {

File: src/ZRCL2.sol

143:     function unlock() external onlyOwner {

154:     function setAllowedSenders(address[] memory _allowedSenders, bool
allow) external onlyOwner {

166:     function setAllowedReceivers(address[] memory _allowedReceivers, bool
allow) external onlyOwner {
```

Remediation:

Consider removing the renounce ownership function.

5.6. Floating pragma is used

Risk Level: Info

Status: Acknowledged

Contracts:

- src/ZRC.sol,
- src/ZRCL2.sol

Description:

Locking the pragma helps ensure that contracts do not accidentally get deployed using a different compiler version.

Remediation:

We recommend locking pragma to a specific Solidity version

```
// SPDX-License-Identifier: UNLICENSED
-pragma solidity ^0.8.20;
+pragma solidity 0.8.26;
```

```
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
import {ERC20Permit} from
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Permit.sol";

/// @title a token contract that locks the transfer function for a period of
time
contract ZRC is Ownable, ERC20Permit {..}
```

5.7. For loop gas optimization

Risk Level: Info

Status: Acknowledged

Contracts:

- src/ZRC.sol,
- src/ZRCL2.sol

Description:

To save some gas while running through an array, it is better to calculate and store length of it locally, for later usage.

Remediation:

```
contract ZRCL2 is IOptimismMintableERC20, Ownable, ERC20Permit, ERC20Votes {
    ...
    constructor(..){
        ...
+       uint256 len = _allowedSenders.length;
-       for (uint256 i = 0; i < _allowedSenders.length; ++i) {
+       for (uint256 i = 0; i < len; ++i) {
        allowedSenders[_allowedSenders[i]] = true;
        emit SetAllowedSenders(_allowedSenders[i], true);
    }
+   len = _allowedReceivers.length;
-   for (uint256 i = 0; i < _allowedReceivers.length; ++i) {
+   for (uint256 i = 0; i < len; ++i) {
        allowedReceivers[_allowedReceivers[i]] = true;
        emit SetAllowedReceivers(_allowedReceivers[i], true);
    }
        ...
    }
    ...
}
```

Same thing can be done in `setAllowedSenders` , `setAllowedReceivers` in both contracts (ZRC, ZRCL2).

5.8. Deployment script uses not the expected private key

Risk Level: Info

Status: Acknowledged

Description:

`.env-template` is a template of an environment, this file has a `PRIVATE_KEY` variable.

So if we will believe an original flow, this `PRIVATE_KEY` from `.env` then should be used during the deployment.

But the deployment script expects to get the private key from a command line, not from `.env`

Remediation:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.26;

import {ZRC} from "../src/ZRC.sol";
import {Script, console} from "forge-std/Script.sol";

contract ZRCScript is Script {
    address BRIDGE = vm.envAddress("L1_BRIDGE");
    address[] allowedSenders;
    address[] allowedReceivers;
    uint256 totalSupply;

    function setUp() public {
        totalSupply = 10_000_000_000 ether;
        // whitelist bridge to enable transfer from the bridge when L2 bridge
to L1
        allowedSenders.push(BRIDGE);
    }

    function run() public {
-         vm.startBroadcast();
+         vm.startBroadcast(vm.envUint("PRIVATE_KEY"));
        // @audit-info it should take private key from .env so
vm.startBroadcast(vm.envUint("PRIVATE_KEY")); should be used,
        ZRC zrc = new ZRC(allowedSenders, allowedReceivers, totalSupply);

        vm.stopBroadcast();
    }
}
```

Same line should be added to `ZRCL2Script` .

5.9. Typo in comments

Risk Level: Info

Status: Fixed in the commit 37cc78f8.

Contracts:

- src/ZRC.sol
- src/ZRCL2.sol

Description:

There is a typo in the comments on ZRC.sol:12, ZRC.sol:14, ZRCL2.sol:18, ZRCL2.sol:20. There is also a missing line in the comment for function _update in ZRCL2.sol.

Remediation:

Consider correcting typos.

```
-/// @notice allowed senders durring locked period
+/// @notice allowed senders during locked period
mapping(address => bool) public allowedSenders;
-/// @notice allowed receiver durring locked period
+/// @notice allowed receiver during locked period
mapping(address => bool) public allowedReceivers;
```

Also consider adding line `allowedReceiver is receiving tokens` to the _update function comment in ZRCL2.sol.

5.10. Require is used instead of safeTransferFrom

Risk Level: Info

Status: Acknowledged

Contracts:

- utils/BatchTransfer.sol

Location: Lines: 17. Function: `batchTransfer`.

Description:

When performing batch transfer of tokens, `require` is used to check whether the transfer is successful or not.

```
require(IERC20(token).transferFrom(msg.sender, recipients[i], amounts[i]), "Transfer failed");
```

Not all tokens return `true` on `transferFrom` , so this may result in a revert for a specific token.

Remediation:

Consider using `safeTransferFrom` from `SafeERC20` instead of `require`.

5.11. Functions lack sanity checks

Risk Level: Info

Status: Acknowledged

Contracts:

- src/ZRCL2.sol

Description:

While getting an input from admin or deployer, it is recommended to validate inputs in order to check that they are not null, etc.

Remediation:

Add check for null input

```
contract ZRCL2 is IOptimismMintableERC20, Ownable, ERC20Permit, ERC20Votes {
    ...
    constructor(...){
+       require(_bridge != address(0), "Input cannot be zero addresss");
+       require(_remoteToken != address(0), "Input cannot be zero addresss");
        ...
        BRIDGE = _bridge;

      REMOTE_TOKEN = _remoteToken;
        ...
    }
    ...
}
```

In addition to that, while adding an address to the allowedReceivers array, the address should not be equal to 0. Otherwise, anyone who has tokens on L2 and is not in allowedSenders will be able to bridge tokens to L1 during the `lock`.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.